Agent-based Computational Demography and Beyond using JAS-mine

Agent-based Computational Demography and Microsimulation using JAS

Matteo Richiardi
Institute for New Economic Thinking at the Oxford Martin School, University of Oxford
Nuffield College, Oxford
Collegio Carlo Alberto, Moncalieri

Ross Richardson
Institute for New Economic Thinking at the Oxford Martin School, University of Oxford
Mathematical Institute, University of Oxford

This draft: January 2015

Abstract:
In this chapter we provide a hands-on guide on how to build a microsimulation using JAS, a Java-based platform that provides unique simulation tools for discrete-event simulations, including both agent-based and microsimulation models. After presenting the rationale for the recent developments of the JAS project and the main architectural choices made, we illustrate a step-by-step implementation of a rich dynamic microsimulation, which includes demographic processes (birth, death, household formation and dissolution) and other life course events (educational choices, labour market participation and employment outcomes).

Acknowledgements:
Matteo Richiardi has benefited from financial support by the European Commission within a Marie Curie IE fellowship, and from support by Piedmont Region (research project: "From work to health and back: The right to a healthy working life in a changing society") and Collegio Carlo Alberto (research project: "Causes, Processes and Consequences of Flexsecurity Reform in the EU: Lessons from Bismarkian Countries"). Michele Sonnessa is the main developer of the JAS library and has contributed to the coding of the microsimulation model.

Keywords: simulation platform, microsimulation, agent-based, software, open-source.

JEL codes: C63, C88

1. Introduction

In this paper, we present the implementation of a dynamic microsimulation with a rich set of demographic processes (birth, death, household formation and dissolution) and other life course events (educational choices, labour market participation and employment outcomes), using the recently upgraded JAS simulation platform (Richiardi and Sonnessa, 2013).

The chapter is meant to provide a step-by-step guide to the development of dynamic microsimulations. From a practical perspective, the model presented here is highly reusable and can be easily modified in order to develop other microsimulation models.[1] This is thanks to the JAS architecture, which envisages a neat separation between data (parameters and coefficients) and code, and a clear distinction between *modelling objects*, which specify the structure of a model and should be the primary concern of a researcher, and *auxiliary objects*, which perform useful tasks such as input-output communication, real-time visualization, etc.

The chapter is structured as follows. Section 2 motivates the need for a unique platform for agent-based and dynamic microsimulations, integrating tools used by both modelling approaches. The section also lists other requirements that were specified for the platform. Section 3 briefly describes the technical solutions that were adopted to meet these requirements. Section 4 presents the general structure of a JAS project. Section 5 describes the specific simulation model implemented. Section 6 illustrates the JAS implementation. Section 7 concludes.

2. Convergence between agent-based and microsimulation models

Microsimulation is a technique used in a large variety of fields to simulate the states and behaviors of different units (individuals, households, populations, etc.) as they evolve in a given environment (a market, a region an institution). The word "dynamic" refers to the fact that the population being simulated is also changing, as opposed to "static" microsimulations (such as tax and benefit simulators) which examine the impact of a policy change on a given population (Li and O'Donoghue, 2012). The modelling of demographic processes is therefore the salient characteristic of dynamic microsimulations.

Agent-based models are also computational models with individuals as the primary object of analysis. They mainly differ from microsimulations for their emphasis on the role of interaction and from explicit departures from the standard assumptions of economic models: rational expectations, perfect knowledge about the environment, infinite computational ability, absence of centralised "top down" coordination devices (Richiardi, 2012).

Agent-based (AB) and microsimulation (MS) models share many features and can be described as belonging to the same class of discrete-event simulations. Indeed, from a mathematical and computational perspective the two approaches are identical. Both AB and MS models are recursive models, where the number and individual states of the agents in the system are evolved by applying a sequence of algorithms to an initial population. As computer-based simulations, they face the problem of reproducing real-life phenomena, many of which are temporally continuous processes, using discrete microprocessors. The abstract representation of a continuous phenomenon in a simulation model requires that all events be presented in discrete terms, hence the label discrete-event simulation.

However, in their historical development AB models and microsimulations have followed different trajectories (Richiardi, 2013): AB models have focused more on theory, while MS models have evolved as more data oriented, with the processes generally specified as probabilistic regression models. As a generalisation, AB models are structural models with a primary concern on *understanding*, while

[1] The model and the supporting documentation can be downloaded from the JAS website (www.jasimulation.org).

microsimulations are reduced-form models (as such, they often focus on one side of a market only), with a primary concern on *forecasting*.[2] However, a trend is currently undergoing towards a convergence of the two approaches, with AB models becoming increasingly empirically oriented, and MS models including more feedback effects (see again Richiardi, 2013). An example of this fruitful integrated approach is the recent field of agent-based computational demography (Billari and Prskawetz, 2003).

The differences in scope and perspective between the two approaches have however impinged on the structure of the computer models used within each community. AB models lead naturally to an explicit object-oriented representation, while MS models are generally built around a database which is evolved forward in time. This has led to the development of simulation toolkits which are specific to each field, as for instance NetLogo (Wilenski 1999), RePast (North et al. 2013), Mason (Luke *et al.* 2005) for AB modelling, and Modgen (Statistics Canada, 2009), LIAM2 (De Menten *et al.* 2012) and JAMSIM (Mannion et al., 2012) for MS modeling – to name just a few.[3]

JAS was created to make the development of "hybrid" AB-MS models easier, and to allow researchers to use the same tools for both approaches, to exploit economies of scales in learning and coding. Its combination of features distinguish it from all the above platforms.

3. The JAS architecture

JAS is an object-oriented Java-based platform for discrete-event simulations. The philosophy of JAS is to favour *clarity*, *transparency* and *flexibility*, even at the expense of brevity and speed of execution. The rationale behind this is the belief that the real bottleneck in agent-based and dynamic microsimulation modelling comes from humans, rather than machines: minimizing modelling time then becomes more important than minimizing computing time. In turn, minimizing modelling time can be achieved by 1) using more transparent and better organized functions, and 2) adhering to a strict modelling discipline that maintains the separation between things that are conceptually separate. This not only speeds up the development of a model, but simplifies modular development and subsequent extensions and modifications.

With respect to the first issue, two key choices were made. First, a strict adherence to the open source paradigm was enforced, which makes it less of a black-box with respect to proprietary software and encourages cooperative development of the platform by the community of users: all functions can be inspected and, if necessary, modified or extended. Second, it was decided not to develop an ad-hoc grammar and syntax –as in Netlogo and LIAM2– but to allow the user to choose from a wide range of classes and interfaces which extend the standard Java language. The JAS libraries therefore provide open tools to "manufacture" a simulation model, making use whenever possible of solutions already available in the software development community (external functions can also be easily added as plug-ins). This also ensures a maximum amount of flexibility in model building.

With respect to the second issue, a clear distinction is made between objects with a modelling content, which specify the structure of the simulation, and objects which perform useful but auxiliary tasks, from enumerating categorical variables to building graphical widgets, from creating filters for the collection of agents to computing aggregate statistics to be saved in the output database. Moreover, a separation is made between code and data, with all parameters and input tables stored either in an input database or in specific MS Excel files. The *regression* package provides tools for simulating outcomes from standard regression models (OLS, probit/logit, multinomial probit/logit): in particular, there is no need to specify the variables

[2] Structural models often include unobservable parameters that help describe individual behaviour at a deep level (say, in terms of utility maximisation); reduced-form models aim more simply at identifying statistical relationships between observable characteristics.

[3] Though it should be noted that the popular NetLogo environment is not object-oriented.

that enter a regression model, as they are directly read from the data files. This greatly facilitates exploration of the parameter space, testing different econometric specifications, and scenario analysis.

From a modelling viewpoint, from version 2 JAS extends the *Model-Observer* paradigm introduced by the Swarm experience (Minar *et al. 199*6) and introduces a new layer in simulation modeling, the *Collector*. The Model deals mainly with specification issues, creating objects, relations between objects, and defining the order of events that take place in the simulation. The Observer allows the user to inspect the simulation in real time and monitor some pre-defined outcome variables as the simulation unfolds. The Collector collects the data and compute the statistics needed both by the simulation objects and for post-mortem analysis of the model outcome, after the simulation has completed. This three-layer methodological protocol allows for extensive re-use of code and facilitates model building, debugging and communication.

As for input/output (I/O) communication, building on the vast number of software solutions available, JAS allows the user to separate data representation and management from the implementation of processes and behavioral algorithms. The management of input data persistence layers and simulation results is performed using standard database management tools, and the platform takes care of the automatic translation of the relational model of the database into the object-oriented simulation framework, through object-relational mapping (ORM).[4] This also allows to separate data creation from data analysis, which is crucial for understanding the behavior of the simulation model. As the statistical analysis of the model output is possibly intensive in computing time, performing it in real time might be an issue, in large-scale applications. A common solution is to limit it to a selected subset of output variables. This, however, requires identifying the output of interest before the simulation is run. If additional computations are required to better understand how the model behaves, the model has to be run again: the bigger the model, the more impractical this solution is. On the other hand, the power of modern relational database management systems (RDBMS) makes it feasible to keep track of a much larger set of variables, for later analysis. Also, the statistical techniques envisaged, and the specific modeler's skills, might suggest the use of external software solutions, without the need to integrate them in the simulation machine. Finally, keeping data analysis conceptually distinct from data production enhances brevity, transparency and clarity of the code.

The main architectural characteristics of JAS are discussed in detail in (Richiardi and Sonnessa, 2013) and on the JAS website (www.jasimulation.org). To summarise, the main features of the platform are:

- a discrete-event simulation engine, allowing for both discrete-time and continuous-time simulation modeling[5];
- a *Model-Collector-Observer* structure (see below);
- interactive (GUI based) batch and multi-run execution modes, the latter allowing for detailed design of experiments (DOE);
- a library implementing a number of different matching methods, to match different lists of agents;

[4] ORM is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

[5] Discrete-event simulations can be organized in two categories, depending on how time is treated. *Continuous-time* simulations break up time into regular time slices (Δt), while the simulator calculates the variation of state variables for all the elements of the simulated model between one point in time and the next. Nothing is known about the order of the events that happen within each time period: discrete events (marriage, job loss, etc.) could have happened at any moment in Δt while inherently continuous events (aging, wealth accumulation, etc.) are often thought to progress linearly between one point in time and the next. By contrast, *discrete-time* simulations are characterized by irregular timeframes that are punctuated by the occurrence of the events. What is modelled is not whether an event occurs or not in the reference period, in a discrete-choice setting, but rather the time elapsed before its occurrence (duration models). Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. Inherently continuous events must then be discretised.

- a library implementing a number of different alignment methods, to force the microsimulation outcomes meeting some exogenous aggregate targets (Li and O'Donoghue, 2014);
- a library implementing a number of common econometric models, from continuous response linear regression models to binomial and multinomial logit and probit models;
- a statistical package based on the *cern.jet* package;
- an embedded H2 database;
- MS Excel I/O communication tools;
- automatic GUI creation for parameters by using Java annotation;
- automatic output database creation;
- automatic agents' sampling and recording;
- powerful probes for real-time statistical analysis and data collection;
- a rich graphical library for real-time plotting of simulation outcomes;
- Eclipse plugin, which allows to create a JAS project in just a few clicks, with template classes organised in the JAS standard package and folder structure;
- Maven version control.

4. The structure of a JAS project

In the JAS architecture, agents are organized and managed by components called *managers*. As already mentioned, there are three types of managers in this architecture: *Model*, *Collector* and *Observer*. *Models* serve to build artificial agents and objects, and to plan the time structure of events. *Collectors* are managers that build data structures and routines to calculate (aggregate) statistics dynamically, and that build the objects used for data persistence. The definition of a *Collector*'s schedule specifies the frequency of statistics updating and agent sampling, and consequent storage in the output database. *Observers* are managers that serve to build graphical widget objects that indicate the state of the simulation in real time, and define the frequency with which to update these objects.

JAS allows multiple Models (and multiple *Collectors* and *Observers*) to run simultaneously, since they share the same scheduler.[6] This allows for the creation of complex structures where agents of different Models can interact. Each *Model* is implemented in a separate Java class that creates the objects and plans the schedule of events for that *Model*. *Model* classes require the implementation of the *SimulationManager* interface, which implies the specification of a *buildObjects* method to build objects and agents, and a *buildSchedule* method for planning the simulation events. Analogously, *Collector* classes must implement the *CollectorManager* interface, and *Observer* classes must implement the *ObserverManager* interface.

When a new JAS project is created using the JAS Eclipse plugin, several packages are created:

- **data**: a package containing the classes that describe the structure of coefficients, parameters and agent population tables contained in the database to be loaded by the ORM. When using Excel files to specify input data, no specific classes need to be included in this package.
- **model**: a package containing the classes that specify the model structure; in particular, it contains the *Model* manager class(es) and the class(es) of agents that populate the simulation.
- **model.enums**: a subpackage containing the definition of the enumerations used (if any).[7]

[6] Technically, the scheduler is a "singleton". In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object.

[7] Enumerations specify a set of predefined values that a property can assume. These values might be categorical (strings, e.g. sex), quantitative (discrete numbers, e.g. age) or even objects with their set of characteristics and properties (e.g. a predefined set of banks to which a firm can be linked). The ORM detects that a value is an enumeration when the

- **experiment**: a package containing the classes that deal with running the simulation experiment(s); it contains, in particular, the *Start* class where the main method and the type of the experiment (interactive vs. batch mode, single run vs. multiple runs) are defined. The package might also contain one or more *Observer* manager classes for online statistics collection and display, and a *MultiRun* class that manages repeated runs for parameter exploration.
- **algorithms**: a package containing classes that implement algorithms for determining events and applying processes to the agents. These implementations, in a cooperative effort of users, are potential candidates to extend the set of standard functions included in the JAS libraries.

In addition to sources, the project also contains two folders for data input-output. The input folder contains data on database inputs in excel or H2 embedded formats. The output folder contains the output of different simulation experiments. At the beginning of each run, JAS creates a sub-folder that is labeled automatically[8] with a copy of the input files plus an empty output database, with the same structure of the input database as defined by the annotations added to the model classes. Coherence between the input database (if any), the output database and the classes representing the agents in the simulation (known as *entity* classes) is guaranteed by the ORM.

By default, JAS executes the simulations in embedded mode: the databases are modified directly by the JDBC driver included in JAS.[9] The standard database uses a H2 database engine. Other databases supporting embedding can be used, such as Microsoft Access, Hypersonic SQL, Apache Derby, etc.

5. The dynamic microsimulation model

The dynamic microsimulation model that we implement is inspired by the *demo07* sample model included in LIAM2.[10] It is not meant to reproduce the actual dynamics of any existing population, and it works only for illustrative purposes. It features a population of 20,200 persons grouped in 14,700 households undergoing a number of demographic and other life course events on an annual basis between the years 2002 to 2061:

- **Birth**: all women aged between 15 and 50 (inclusive) in any simulation year can give birth to a child, with a probability which is year- and age-specific and is reported in the file *p_birth.xls*.
- **Education**: education (lower secondary, upper secondary or tertiary) is predetermined at birth. Individuals exit lower secondary education at age 16, upper secondary education at age 19, and tertiary education at age 24.
- **Exit from parental home**: individuals aged 24 or over who are not yet married leave their parental home to set up a new household.
- **Marriage**: all individuals aged 18 or over whose civil state is either single, divorced or widowed, are eligible for getting married in any given simulation year. The probability of marriage depends on age, gender and civil state, and is stored in the *al_p_mmkt.xls* file. Given these probabilities, a subset of the unmarried population is sampled and those chosen are entered into the matching algorithm. Actual matching involves ordering all the females first; then starting with the top ranked female, all

property is declared with the annotation *@Enumerated* (see below). Through enumerations the ORM automatically manages reading/writing operations in both text and numerical format.

[8] The folder name can be modified dynamically through labels set by the user.

[9] A JDBC driver is a software component enabling a Java application to interact with a database.

[10] The model differs from the LIAM2 version in that it collapses the work states of unemployment and inactivity into a single non-employment state. This is done by removing the unemployment module from the corresponding LIAM2 simulation, with everything else staying the same. The change is motivated by the fact that the distinction between unemployment and inactivity was implemented in a very unnatural way in LIAM2, and did not affect any subsequent choice on the part of the agents.

males are ordered and the best available male is matched. Then for the second ranked female, the remaining males are ordered and the best available male is matched, and so on until no more matches can be made (because there are either no more males or females to match). Females are ordered according to their age difference (in absolute value) with respect to the average age in the pool of females to be matched, |*age*-mean(*age*)|: females with an age closer to average "choose" first, while older or younger females "choose" later. For each female, males are ranked by looking at how their age and work status compares with the female's age and work status: regression coefficients are stored in the *reg_mariage_score.xls* file.

- **Divorce**: divorce probability depends on age difference between the partners, elapsed marriage duration, number of children and work status of both partners: regression coefficients are stored in the *reg_divorce.xls* file.
- **Employment**: all individuals who are of working age (males: between 15 and 65; females: between 15 and 61) and whose previous work state was neither student nor retired are considered to be available to work. Conditional on this, individuals are employed with a probability which depends on age, lagged work state (either employed, unemployed or inactive), gender and marital status: regression coefficients are stored in the *reg_inwork*.xls file . The model does not distinguish between unemployment and non-employment.[11]
- **Death**: death is also a probabilistic event, with year- and age-specific death probabilities contained in the files *p_death_m.xls* and *p_death_f.xls*, for males and females respectively.

The divorce and employment processes are subject to alignment. This is a technique widely used in (dynamic) microsimulation modeling to ensure that the simulated totals conform to some exogenously specified targets, or aggregate projections (Baekgaard, 2002; Klevmarken, 2002, Li and O'Donoghue, 2014). Alignment is a way to incorporate additional information which is not available in the estimation data. The underlying assumption is that the microsimulation model is a poor(er) model of the aggregate, but a good model of individual heterogeneity: by forcing the microsimulation outcomes to match the targets in a way that is as least distortive as possible, the microsimulation model is left with the task of distributing the totals in the population. In general, the above assumption is very dangerous and unwarranted, and alignment should be looked at with great suspicion. Here, we do not dig into this discussion, and simply replicate the LIAM2 implementation, using the JAS alignment libraries. Alignment targets (aggregate frequencies) are stored in the *p_divorce.xls* and *p_inwork.xls* respectively for divorce and employment.

One important thing to note is that the processes to be aligned are executed at an individual level, while alignment always takes place at the population level. That is, individual outcomes or probabilities are determined for each individual based on the chosen econometric specification and the estimated coefficients. This in general leads to a mismatch between the simulated (provisional) totals and the aggregate targets, which can of course be assessed only at the population level. The alignment algorithm then directly modifies the individual outcomes or probabilities of occurrence.

The specific algorithm used in the LIAM2 implementation is called "Sorting by the difference between the predicted probability and a random number" (SBD, see Li and O'Donoghue, 2014), and – though quite common in the microsimulation literature – can be criticised over many theoretical grounds (see Stephensen, 2014). The JAS *alignment* library implements it for completeness, though its use is deprecated. Here we use it to remain as close as possible to the original LIAM2 version (the reader does not need to understand how precisely it works).

[11] According to the International Labour Organization (ILO), the unemployed comprise all persons above a specified age who during the reference period were: (i) without work, that is, were not in paid employment or self employment; (ii) currently available for work, that is, for paid employment or self-employment; and (iii) seeking work, that is, had taken specific steps in a specified recent period to seek paid employment or self-employment. Non-employment includes inactive people of working age (either because they do not wish to work or because they are not actively seeking work).

6. The JAS implementation

The JAS class structure of the *demo07* model is organised as in Table 1.

Package	Class
algorithms	MapAgeSearch
data	Parameters
data.filters	ActiveMultiFilter FemaleFilter FemaleToCoupleFilter FemaleToDivorce MaleFilter MaleToCoupleFilter
experiment	PersonsCollector PersonsObserver StartPersons
model	Household Person PersonsModel
model.enums	CivilState Education Gender WorkState

Table 1. Class structure

The core of the simulation lies in the *model* package, which contains the classes *PersonModel*, *Person* and *Household*. The *experiment* package contains the *StartPerson* class that specifies to run the simulation in interactive mode, the *PersonsCollector* class that collects all the statistics needed to monitor the simulation and updates the output database, and the *PersonsObserver* class that creates and manages the graphical object for runtime monitoring. Parameters and coefficients are loaded into the *Parameters* class in the *data* package. All filters used to filter collections are grouped in the *data.filters* subpackage. The categories used for gender, educational levels, civil state and work state are stored as *Enums* in the *model.enums* subpackage. Finally the *algorithms* package contains classes that perform technical tasks (in the example, *MapAgeSearch* searches age- and gender- specific values in a map of coefficients, with age and gender as keys). Classes in the *algorithms* package are meant to be of general use beyond the specific model being developed, and are candidates for inclusion in the core libraries in future releases of JAS.

The project is organised in the standard JAS folder structure, as described in Table 2.

Folder	Files	Notes
input	p_birth.xls	Birth probabilities, by age and (simulated) year
	p_death_f.xls	Death probabilities, by age and (simulated) year, for females
	p_death_m.xls	Death probabilities, by age and (simulated) year, for males
	al_p_mmkt.xls	Marriage probability, by age group, gender and civil state
	p_divorce.xls	Divorce probability, by age group and (simulated) year
	coeff_marriage_fit.xls	Marriage score coefficients: determine how well a specific male fits a given female
	coeff_divorce.xls	Divorce coefficients: determine the (unaligned) probability of divorcing
	coeff_inwork.xls	Employment coefficients: determine the (unaligned) probability of being employed
	input.h2	Initial population
output	(empty)	
src	(all java classes)	See table 1
target	(compiled classes, JARs etc.)	
libs	(external libraries, empty)	
(root)	pom.xml	

Table 2. File structure

The Java classes are contained in the *src* folder. The *input* folder contains the MS Excel parameter files and the initial population, stored as an h2 database (input.h2). The *output* folder is initially empty. For each new simulation experiment, a new subfolder is created with the appropriate time stamp, so as to uniquely identify the experiment (e.g. 20141218151116, for experiments initiated on the 18th December 2014, at 16 seconds after 3.11pm). The subfolder contains a copy of all the input files (in the *input* directory) and an output database (out.h2, in the *database* directory).

The *target* and *lib* folders contain technical content of no immediate interest to the modeller. The root folder also contains a *pom* (project object model) file, which contains information on the JAS version used for the project. Apache Maven, an open source software project management and comprehension tool uses this information to manage all the project's build, reporting and documentation. In particular, by specifying in the *pom* file the desired release for each library used (including the JAS libraries), Maven automatically downloads the relevant libraries from the appropriate repositories.[12]

6.1 Parameters

As JAS supports a clear distinction between modelling classes and data structures, parameters are loaded and stored in a specific class, *Parameters.java*. The class makes use of the *ExcelAssistant.loadCoefficientMap*() method to read the parameters from MS Excel files: this requires to specify a .xls file, a sheet name, the number of key columns and the number of value columns.[13] Parameters

[12] This implies that each JAS project has its own copy of all the libraries used, ensuring that the project is self-contained and that it keeps working exactly as intended even when new versions of the libraries are released (and even if backward compatibility is not respected).

[13] It is also possible to load the parameters from a table in the input database. See the online documentation for further details.

are then stored in *MultiKeyCoefficientMap* objects, which are basically standard Java maps with multiple keys (Box 1).

```
public static void loadParameters() {

        // probabilities
        pBirth = ExcelAssistant.loadCoefficientMap("input/p_birth.xls", "Sheet1", 1, 59);
        pDeathM = ExcelAssistant.loadCoefficientMap("input/p_death_m.xls", "Sheet1", 1, 59);
        pDeathF = ExcelAssistant.loadCoefficientMap("input/p_death_f.xls", "Sheet1", 1, 59);
        pMmkt = ExcelAssistant.loadCoefficientMap("input/al_p_mmkt.xls", "Sheet1", 3, 4);
        pDivorce = ExcelAssistant.loadCoefficientMap("input/p_divorce.xls", "Sheet1", 2, 59);
        pInWork = ExcelAssistant.loadCoefficientMap("input/p_inwork.xls", "Sheet1", 3, 59);

        // coefficients
        coeffMarriageFit = ExcelAssistant.loadCoefficientMap(
                "input/reg_marriage_fit.xls", "Sheet1", 0, 9);
        coeffDivorce = ExcelAssistant.loadCoefficientMap(
                "input/reg_divorce.xls", "Sheet1", 0, 6);
        coeffInWork = ExcelAssistant.loadCoefficientMap(
                "input/reg_inwork.xls", "Sheet1", 2, 5);

        // definition of regression models
        regMarriageFit = LinearRegression(coeffMarriageFit);
        regDivorce = ProbitRegression(coeffDivorce);
        regInWork = ProbitRegression(coeffInWork);

}
```

Box 1. The *Parameters.loadParameters*() method.

There are two types of parameters in *demo07*: probabilities and regression coefficients.

Birth and death probabilities have one key (age), while the value columns refer to different simulation years: birth and death probabilities are therefore age- and year-specific.

Divorce probabilities have two keys (the lower and upper bounds defining age groups), while value columns refer again to different simulation years: divorce probabilities are therefore age group- and year-specific.

Marriage and employment probabilities have three keys (the lower and upper bounds defining age groups and gender), while value columns refer to different civil states: marriage and employment probabilities are therefore age group- and civil state-specific.

Table 3 shows how the p_birth.xls file looks like.

	Simulation year		
age	**2002**	**…**	**2061**
15	0.00068	…	0.00075
16	0.00186	...	0.00167
17	0.00595	…	0.00557
18	0.01141	…	0.01158
…			
50	0.00010	…	0.00021

Table 3. Extract from the *p_birth.xls* file

Regression coefficients have one key which is the variable name, and a corresponding value with the estimated coefficient. They might have additional keys, as in *reg_inwork.xls*, if the coefficients are differentiated by some other variables (gender and employment state, in this example). Table 4 shows how the *reg_inwork.xls* file looks like.

vble name	gender	work state	coeff
age	Male	Employed	-0.19660
married	Male	Employed	0.18928
const	Male	Employed	3.55461
…			
age	Male	NotEmployed	0.97809
const	Male	NotEmployed	-12.39108
…			
age	Female	Employed	-0.27405
married	Female	Employed	-0.09068
const	Female	Employed	3.64871
…			
age	Female	NotEmployed	0.82176
married	Female	NotEmployed	-0.55910
constant	Female	NotEmployed	-10.48043
…			

Table 4 Extract from the *coeff_inwork.xls* file

The appropriate regression models are then defined based on the regression coefficients.

6.2 The PersonsModel class

6.2.1 Objects

The Model extends the *AbstractSimulationManager* class. This requires implementing the *buildObjects* and the *buildSchedule* methods. The *buildObjects* method contains the instructions to create all the agents and the objects that represent the virtual environment for model execution (Box 2).[14] In *demo07*, this involves loading the parameters for the simulation and the initial population, made of persons and households. Three other methods complete the simulation setup: *initializeNonDatabaseAttributes*() initializes attributes that do not appear in the input database, such as the education level; *addPersonsToHouseholds*() registers household members, and *cleanInitialPopulation*() checks the internal consistency of the initial population and removes errors, making sure that all marriage partnerships are bilateral, that all partners belong to the same household, and that no empty households exist.[15]

[14] The *@Override* annotation is used by the Java interpreter to ensure that the programmer is aware that the method declared is overriding the same method declared in the superclass.
[15] This method is absent in the LIAM2 implementation, which does not get rid of all the errors in the initial database.

```
@Override
public void buildObjects() {

        Parameters.loadParameters();
        System.out.println("Parameters loaded.");

        persons = (List<Person>) DatabaseUtils.loadTable(Person.class);
        households = (List<Household>) DatabaseUtils.loadTable(Household.class);
        System.out.println("Initial population loaded.");


        initializeNonDatabaseAttributes();
        addPersonsToHouseholds();
        cleanInitialPopulation();

}
```

Box 2. The *PersonsModel.buildObjects*() method.

As we have seen, the general rule is that parameters should not be hard-coded in the simulation. The only exception is with *control parameters* that can be changed from the GUI before the simulation starts or while the simulation is running in order to experiment with the model behavior in interactive mode. Control parameters are properties of a simulation, they are annotated with *@ModelParameter* and are automatically loaded into the JAS GUI. In *demo07* there are just three such parameters, as described in Box 3.

```
@ModelParameter(description="Simulation begins at year [valid range 2002-2060]")
private Integer startYear = 2002;

@ModelParameter(description="Simulation ends at year [valid range 2003-2061]")
private Integer endYear = 2061;

@ModelParameter(description="Retirement age for women")
private Integer wemra = 61;
```

Box 3. PersonsModel: scenario parameters.

6.2.2 Schedule

The *buildSchedule*() method contains the plan of events for the simulation. Events are planned based on a discrete event simulation paradigm. This means that events can be scheduled dynamically at specific points in time. The frequency of repetition of an event can be specified in the case of periodic events. An event can be created and managed by the simulation engine (a system event e.g. terminating the simulation), it can be sent to all the components of a collection or list of agents or it can be sent to a specific object/instance. Events can be grouped together if they share the same schedule.

In *demo07*, all events are scheduled right from the beginning of the simulation (no dynamic scheduling), and occur on a yearly basis. They are grouped in an *EventGroup* called *s*, which is scheduled at every simulation period starting at 0 with the *schedule*(*Event event, long atTime, int withLoop*) method:

```
getEngine().getEventList().schedule(s, 0, 1);
```

The events of *demo07* are typically directed to a collection of objects – persons or households – and are inserted into an EventGroup with the instruction

```
s.addCollectionEvent(Object object, [some action the object must perform]);
```

The actions to be performed can be specified in two ways. The simplest is to use Java reflection and simply specify the object's method name to be invoked. For instance, asking all persons to perform the *aging*() method would require the instruction:

```
s.addCollectionEvent(persons, "ageing");
```

Use of Java reflection however is quite slow. A better approach is to use the *EventListener* interface. When an object implements this interface, it must define an *onEvent*() method that will receive specific enumerations to be interpreted. We will describe how the *Person* and *Household* classes implement the *onEvent*() method in due time. For now, we simply note that by using the *EventListener* interface, the scheduling of the *ageing*() method becomes:

```
s.addCollectionEvent(persons, Person.Processes.Ageing);
```

By default, the broadcasting of an event to a collection of objects is performed in *safe mode* (read only), and does not allow the concurrent modification of the collection itself. This is not a problem with the *ageing*() process, as ageing *per se* does not entail any modification in the list of persons, that is, it does not add nor remove anyone. This is not true with other processes, like *birth*() or *death*(). In order to allow the collection to be changed while iterated by the simulation engine, this feature has to be switched off, as in

```
s.addCollectionEvent(persons, Person.Processes.Death, false);
```

The last argument specifies that the collection is subject to changes while iterated, and the JAS engine treats it accordingly.

The order of the events in the simulation follows the original LIAM2 implementation and is specified as in Box 4: there is a first set of demographic events (ageing, death, birth, marriage, exit from parental home, divorce, household composition) and then a set of events that define the work status (whether in education, retired, other non-employed, or employed).

```
@Override
public void buildSchedule() {

        EventGroup s = new EventGroup();
        // 1: Ageing
        s.addCollectionEvent(persons, Person.Processes.Ageing);

        // 2: Death
        s.addCollectionEvent(persons, Person.Processes.Death, false);

        // 3: Birth
        s.addCollectionEvent(persons, Person.Processes.Birth, false);

        // 4: Marriage
        s.addCollectionEvent(persons, Person.Processes.ToCouple, false);
        s.addEvent(this,  Processes.MarriageMatching);

        // 5: Exit from parental home
        s.addCollectionEvent(persons, Person.Processes.GetALife, false);

        // 6: Divorce
        s.addEvent(this,  Processes.DivorceAlignment);
        s.addCollectionEvent(persons, Person.Processes.Divorce, false);

        // 7: Household composition
        // (for reporting only: household composition is updated whenever needed throughout
        // the simulation)
        s.addCollectionEvent(households, Household.Processes.HouseholdComposition);

        // 8: Education
        s.addCollectionEvent(persons, Person.Processes.InEducation, false);
```

```
            // 9: Work
            s.addEvent(this, Processes.InWorkAlignment);

            getEngine().getEventList().schedule(s, 0, 1);

            getEngine().getEventList().schedule(
               new SingleTargetEvent(this, Processes.Stop), endYear - startYear );

    }
```

Box 4. The *PersonsModel.buildSchedule*() method.

Marriage is performed in two steps. First, a subset of suitable males and females are selected for matching by invoking the method *Person.toCouple*()[16]; then, matching takes place. As we have seen, matching uses a "centralised" algorithm and is therefore performed by the Model itself. Consequently, this event is a single target event, rather than a collection event, and is inserted into our EventGroup *s* with the instruction

```
s.addEvent(this,  Processes.MarriageMatching);
```

Similarly, the divorce and work events are subject to alignment and are managed directly by the Model, with the methods *divorceAlignment*() and *inWorkAlignment*(), though divorce also requires some actions taken by the individuals themselves – with the method *divorce*() – after they have been selected to divorce. *householdComposition*() is the only method which is directed to the collection of households. It simply updates the number of adults and children in each household for reporting purposes. A final single target event is scheduled for the last year of the simulation with the method *schedule(Event event, long atTime*): its target is the *Model* itself and brings the simulation to a halt:

```
getEngine().getEventList().schedule(
            new  SingleTargetEvent(this, Processes.Stop), endYear - startYear);
```

6.2.3 The EventListener interface

Since the Model performs actions during the simulation, as with the *Person* and *Household* classes, it implements the *EventListener* interface. This requires first to enumerate all the actions that the Model is supposed to perform (this is done by defining the specific *enum Processes*), and then to specify the method *onEvent*() – see Box 5.

```
public enum Processes {
            MarriageMatching,
            DivorceAlignment,
            InWorkAlignment,
            Stop;
    }

    @Override
    public void onEvent(Enum<?> type) {
            switch ((Processes) type) {
            case DivorceAlignment:
                   divorceAlignment();
                   break;
            case InWorkAlignment:
                   inWorkAlignment();
                   break;
            case MarriageMatching:
                   marriageMatching();
                   break;
            case Stop:
```

[16] See section 6.3 below.

```
                        log.info("Model completed");
                        getEngine().pause();
                        break;
                }
        }
```

Box 5. Implementation of the *EventListener* interface in *PersonsModel*.

We now dig into the matching and alignment methods performed by the *Model*.

6.2.4 The matching algorithm

Prior to matching, a sample of the population to marry at this time is determined randomly using the *Person.toCouple*() method. Subsequently, matching involves first ordering all the females; then, for each female starting from the top of the ranking, all males are ordered and the most suitable male is matched. This continues until there are either no more females or males to match. Females are ordered according to their age difference (in absolute value) with respect to the average age in the pool of females to be matched, |*age*-mean(*age*)|; the female whose age is closest to the average is ranked first. To compute this ranking, the average age of the subset of females selected for matching is required. There are a number of ways to perform this computation, which is preliminary to the application of the matching algorithm. The one that is implemented in *demo07* makes use of Java closures (Box 6).[17]

```
        final AverageClosure averageAge = new AverageClosure() {
                @Override
                public void execute(Object input) {
                        add( ((Person) input).getAge() );
                }
        };

        Aggregate.applyToFilter(getPersons(), new FemaleToCoupleFilter(), averageAge);
```

Box 6. Computing the average age for the would be spouses in *PersonsModel.MarriageMatching*().

The JAS *collection* package defines an *AverageClosure* as a closure that receives values from objects as an input and returns the mean of these values as an output. Here, it is used to compute the average age of a given set of persons. The set is defined by applying the *FemaleToCouple* filter to the list of all persons, with the instruction

```
Aggregate.applyToFilter(getPersons(), new FemaleToCoupleFilter(), averageAge);
```

The *averageAge* closure now contains the average age of all filtered females. In turn, the *FemaleToCouple* filter simply selects the female persons who have the *toCouple* flag switched on (Box 7).

```
public class FemaleToCoupleFilter implements Predicate {

        @Override
        public boolean evaluate(Object object) {
                Person agent = (Person) object;
                return (agent.getGender().equals(Gender.Female) && agent.getToCouple());
        }

}
```

Box 7. The *FemaleToCouple* filter.

[17] Technically, a *closure* is a function that refers to free variables in their lexical context. A free variable is an identifier (the identity of the person which is included in the evaluation set, in our example) that has a definition outside the closure: it is not defined by the closure, but it is used by the closure. In other words, these free variables inside the closure have the same meaning they would have had outside the closure.

Having the filters specified as separate classes, grouped in the separate package *data.filters*, might look cumbersome at first (and there are other ways to do this, see the online documentation) but allows to keep the core code clean while using the standard Apache Predicate approach to filtering – remember that the JAS approach supports the use of existing software solutions whenever possible, and envisages to keep the specificities of the JAS libraries to a minimum in order to minimise the "black box" feeling of many simulation platforms.

Matching is then performed, following the LIAM2 implementation, by making use of a simple one-way matching procedure (the agents in one collection – females in our example – choose, while the agents in the other collection – males – remain passive) implemented in the *SimpleMatching* class:

```
matching(collection1, filter1, comparator1, collection2, filter2, matchingScoreClosure,
        matchingClosure);
```

and it is invoked as

```
SimpleMatching.getInstance().matching(...);
```

The matching method requires 7 arguments:

1. **collection1**: the first collection (e.g. all individuals in the population);
2. **filter1**: a filter to be applied to the first collection (e.g. all females with the *toCouple* flag on);
3. **comparator1**: a comparator to sort the filtered collection, which determines the order that the agents in the filtered collection will be matched.
4. **collection2**: the second collection, which can be the same as *collection1* (e.g. all individuals in the population) or a different one; the two collections do not need to have the same size;
5. **filter2**: a filter to be applied to the second collection (e.g. all males with the *toCouple* flag on);
6. **matchingScoreClosure**: a piece of code that assigns, for every element of the filtered *collection1*, a double value to each element of the filtered *collection2*, as a measure of the quality of the match between every pair;
7. **matchingClosure**: a piece of code that determines what to do upon matching.

As in the computation of the average age, the use of closures – which are relatively new to the Java language – allows a great simplification of the code. While it is not required that the user knows about closures, it is interesting to understand why they are so useful. In the example, suppose that the females in the population are sorted according to some criterion, for example beauty: the prettiest woman is the first to choose a partner, the second prettiest is the second to choose etc. The *matchingScoreClosure* sorts all possible mates according to some other criterion, for example wealth. Hence, the prettiest woman gets the richest man, the second prettiest gets the second richest, etc. In such a case, a comparator would suffice to order the males in the population, as the ranking is the same irrespective of the female who is evaluating them. But suppose now that the attractiveness of a man depends on the age differential between himself and the potential partner: in such a case, the ranking is specific to each woman in the population. A simple comparator would still do the job, but the comparator should be able to access the identity of the woman who is making the evaluation as an argument, which requires a lot of not-so-straightforward coding. Closures allow to bypass this technical requirement because they can pass a functionality as an argument to another method; in other words, they treat functionality as method argument, or code as data.

Closures in the *matching*() method are easier used than explained: the 7 arguments are listed in Box 8.

```
SimpleMatching.getInstance().matching(

        // collection1: the whole population
        getPersons(),

        // filter1:
        new FemaleToCoupleFilter(),

        // comparator1: a comparator that
        // assigns priority to the individual that has a lower difficulty in matching
        // (this is determined by an individual's age in relation to the average)
        new Comparator<Person>() {

                @Override
                public int compare(Person female1, Person female2) {
                        return (int) (Math.abs(o1.getAge() - averageAge.getAverage()) -
                                Math.abs(o2.getAge() - averageAge.getAverage()));

                }
        },

        // collection2: same as collection1
        getPersons(),

        // filter2:
        new MaleToCoupleFilter(),

        // MatchingScoreClosure: a closure that, given a specific female,
        // computes for every male in the population a matching score
        new MatchingScoreClosure<Person>() {

                @Override
                public Double getValue(Person female, Person male) {
                        return male.getMarriageScore(female);

                }
        },

        // matchingClosure: a closure that creates a link between a specific
        // female and a specific male, and sets up a new household.
        new MatchingClosure<Person>() {

                @Override
                public void match(Person female, Person male) {

                        female.marry(male);
                        male.marry(female);

                }
        }
);
```

Box 8. The matching algorithm in *PersonsModel.MarriageMatching*().

6.2.5 Alignment

Alignment involves comparing the *provisional* outcomes of the simulation with some external aggregate targets, and then modifying the simulation outcomes in order to match the external totals. We show how this is implemented in *demo07* by looking at the *divorceAlignment*() method; the *inWork*() alignment method works similarly. When it comes to divorce, as in *marriageMatching*(), the focus is on females: males are passive recipients of their partners' choices. Different targets are specified for different age groups and simulated years; as we have seen, these are read from the file *p_divorce.xls* and stored into the *MultiKeyCoefficientMap pDivorce* in the *Parameters* class. The *divorceAlignment*() method works cell by cell, that is, it aligns each age group of the population to its year-specific target: this means that the alignment algorithm is applied once for every age group (as defined in the .xls parameter file). The structure of the method is therefore as follows:

- For each age group: do alignment:

- Read target.
- Select the relevant subgroup of married females.
- Compute, for each of the selected females, a probability to divorce.
- Select the couples that divorce by applying the SBD algorithm: females are ranked according to the difference between their divorce probability and a random number uniformly distributed between 0 and 1; then, the number of couples equal to the target are selected to divorce by starting with the top ranked female and going down the ranks until the target number is reached.[18]

The *MultiKeyCoefficientMap pDivorce*, which contains the targets, has a three dimensional key: the lower and upper bounds for the age group, and the year of the simulation. The age-group-specific and year-specific targets are read with the instruction reported in Box 9.

```
    MultiKeyCoefficientMap map = Parameters.getpDivorce();

    for (MapIterator iterator = map.mapIterator(); iterator.hasNext();) {

            iterator.next();
            MultiKey mk = (MultiKey) iterator.getKey();
            int ageFrom = (Integer) mk.getKey(0);
            int ageTo = (Integer) mk.getKey(1);
            double divorceTarget = ( (Number) map.getValue(
                    ageFrom,
                    ageTo,
                    getStartYear() + SimulationEngine.getInstance().getTime())).doubleValue();
            […]
}
```

Box 9. *PersonsModel.divorceAlignment*(): reading the targets.

The alignment methods require 4 arguments:

1. **collection**: a collection of individuals whose outcome or probability of an event has to be aligned (e.g. all the population);
2. **filter**: a filter to be applied to the collection (e.g. all females selected to divorce);
3. **alignmentProbabilityClosure** or **alignmentOutcomeClosure**: a piece of code that i) computes for each element of the filtered collection a probability for the event (in the case that the alignment method is aligning probabilities, as in the SBD algorithm) or an outcome (in the case that the alignment method is aligning outcomes), and ii) applies to each element of the filtered collection the specific instructions coming from the alignment method used;
4. **targetShare** or **targetNumber**: the share or number of elements in the filtered collection that are expected to experience the transition.

Box 10 shows how the alignment method is implemented in *demo07*.

```
    new SBDAlignment<Person>().align(

            // collection:
            getPersons(),

            // filter:
            new FemaleToDivorce(ageFrom, ageTo),
```

[18] The ranking involves a stochastic component (the random number that is subtracted from the divorce probability score) in order to give individuals with a low predicted probability some chance to experience the event. As we have already noted, the SBD algorithm is quite distortive and its use is deprecated in JAS; it is employed here only for consistency with the LIAM2 implementation.

```
        // alignmentProbabilityClosure:
        new AlignmentProbabilityClosure<Person>() {

                // i) compute the probability of divorce
                @Override
                public double getProbability(Person agent) {
                        return agent.computeDivorceProb();
                }

                // ii) determine what to do with the aligned probabilities
                @Override
                public void align(Person agent, double alignedProbability) {

                        boolean divorce = RegressionUtils.event(
                                alignedProbability,
                                SimulationEngine.getRnd()
                        );

                        agent.setToDivorce(divorce);

                }
        },

        // targetShare:
        divorceTarget
);
```

Box 10. *PersonsModel.divorceAlignment*(): applying the SBD alignment algorithm.

6.3 The Person class

6.3.1 Entities

The *Person* class is an *Entity* class, as specified by the *@Entity* annotation:

```
@Entity
public class Person implements Comparable<Person>, EventListener {
        […]
}
```

This implies that the class is linked to a table in the database with the same name, and that all properties which are not annotated as *@Transient* are persisted in the database, when the simulation output is dumped. Entity classes must specify a *PanelEntityKey* (annotated as *@Id*), which is a three-dimensional object which identifies the agent id, the simulation time and the simulation run. These three keys uniquely identify each record in the database:

```
@Id
private PanelEntityKey key;
```

The ORM expects that the field names in the database are the same as the property names in the Java class, except when a different name is specified as in

```
@Column(name="dur_in_couple")
private Integer durationInCouple;
```

Enumerations can be interpreted by the ORM both as a string and as ordinal values (0 for the first *enum*, 1 for the second, etc.), depending on how they are annotated:

```
@Enumerated(EnumType.ORDINAL)
private WorkState workState;
```

6.3.2 Methods

The *Person* class implements the *EventListener* interface and is therefore able to be activated by the scheduler with the *onEvent*() method. The calls that a *Person* is able to respond to – enumerated in a specific *Enum* (Box 11) – are:

- **Ageing**: age and marriage duration are increased; work status is set to retired if retirement age is reached.
- **Death**: an age-, gender- and time-specific death probability is read from the *MultiKeyCoefficientMaps pDeathM* and *pDeathF* stored in the *Parameters* class; this probability is then compared with a uniformly distributed random number between 0 and 1 to determine the occurrence of the event:

  ```
  RegressionUtils.event(deathProbability)
  ```

  If death occurs, the partner's status is updated to widow and the person is removed from all the lists (that is, from his/her household and from the model).
- **Birth** (applied to all females aged between 15 and 50 included). An age- and time-specific probability of having a baby is read from the *MultiKeyCoefficientMap pBirth* stored in the *Parameters* class; then the occurrence of the event is determined in a similar fashion to the *death*() process. No multiple births such as twins can occur. Newborns are given a potential educational level that will be reached with certainty. Following the LIAM2 implementation, the person is assumed to be a student until completion of their studies (at age 16 for lower secondary education, 19 for upper secondary education, and 24 for tertiary education).
- **ToCouple** (applied to all individuals aged between 18 and 90 included who are not married). This method reads a probability of forming a partnership and determines whether the Boolean flag *toCouple* is switched on (i.e. set to true), to be used by the matching algorithm.
- **GetALife** (leave parental home): a new household is created if the individual is aged 24 or over, unmarried and still leaving in the parental household.
- **Divorce**: after divorce is decided by the Model's alignment method, links are broken and new households are created.
- **InEducation**: this method determines whether an individual is still in education or exits education and enters the labour market as unemployed.

```
public enum Processes {
        Ageing,
        Death,
        Birth,
        ToCouple,
        Divorce,
        GetALife,
        InEducation;
}
```

Box 11. The *Person.Processes Enum*, defining the processes a *Person* undertakes when activated by the scheduler.

Other methods of the *Person* class are:

- ***getMarriageScore()***: computes the score of each male in the marriage pool, for a given female, based on a linear regression model; it is used by the matching method in the *Model*.
- ***marry***(): creates a link between the two partners and sets up a new household; it is used by the matching method in the *Model*.
- ***computeDivorceProb*()**: computes the divorce probability, based on a probit regression model; it is used by the alignment method in the *Model*.

- ***computeWorkProb*()**: computes the employment probability, based on a probit regression model; it is used by the alignment method in the *Model*.

Simulation of outcomes or probabilities based on regression models is straightforward:

```
marriageFit = Parameters.getRegMarriageFit().getScore(this);
divorceProb = Parameters.getRegDivorce().getProbability(this);
workProb = Parameters.getRegInWork().getProbability(this);
```

If the specification of the model is changed by adding or removing covariates, or if new estimates become available, nothing has to be changed in the code.

6.4 The Household class

This class contains a list of all household members and is able to count the number of adults and children in the household. It is defined as an *Entity* class and is therefore linked to a table with the same name in the database.

6.5 The PersonsCollector class

The Collector collects statistics and manages the persistence of the simulation outputs on the database. It extends the *AbstractSimulationCollectorManager* interface and requires, similarly to the *Model*, the implementation of a *buildObjects*() method and a *buildSchedule*() method.

The *buildObjects*() method creates several *CrossSection* objects, which collect specific values from each individual in the population (Box 12).

```
@Override
public void buildObjects() {

        final PersonsModel model = (PersonsModel) getManager();
        ageCS = new CrossSection.Integer(model.getPersons(), Person.class, "age", false);
        nonEmploymentCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getNonEmployed", true);
        employmentCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getEmployed", true);
        retiredCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getRetired", true);
        inEducationCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getStudent", true);
        lowEducationCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getLowEducation", true);
        midEducationCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getMidEducation", true);
        highEducationCS = new CrossSection.Integer(model.getPersons(), Person.class,
                        "getHighEducation", true);

}
```

Box 12. The *PersonsCollector.buildObjects*() method.

The Collector's schedule is made up of two events only, that take place at every simulation period: the *CrossSections* are updated, and the persons and households are persisted in the database (Box 13).

```
@Override
public void buildSchedule() {

        EventGroup s = new EventGroup();
```

```
        s.addEvent(this, Processes.Update);
        s.addEvent(this, Processes.DumpInfo);

        getEngine().getEventList().schedule(s, 0, 1);
}
```

Box 13. The PersonsCollector.*buildSchedule*() method.

Updating the CrossSection objects only involves simple instructions such as

```
ageCS.updateSource();
```

Similarly, dumping the simulation outputs only requires

```
DatabaseUtils.snap( ( (PersonsModel) getManager()).getPersons() );
DatabaseUtils.snap( ( (PersonsModel) getManager()).getHouseholds() );
```

6.6 The PersonsObserver class

The *PersonsObserver* builds the graphical objects that allow the monitoring and inspection of the simulation outcome in real time. It extends the *AbstractSimulationObserverManager* interface and, similarly to the other simulation managers (the *Model* and the *Collector*), requires the implementation of a *buildObjects*() method and a *buildSchedule*() method.

The *buildObjects*() method creates three plot. The first one (*agePlotter*) depicts the evolution of the average age of the simulated population: it takes the *ageCS CrossSection* from the *Collector*, with information on the age of each individual, and computes its mean (by creating a *MeanArrayFunction* object). Similarly, the *workPlotter* plots the frequency of students, retired, other non-employed and employed individuals in the population, and the *eduPlotter* plots the share of individuals with each educational level (Box 14).

```
@Override
public void buildObjects() {
        final PersonsCollector collector = (PersonsCollector) getCollectorManager();

    agePlotter = new TimeSeriesSimulationPlotter("Age", "age");
    agePlotter.addSeries("avg", new MeanArrayFunction(collector.getAgeCS()));
    GuiUtils.addWindow(agePlotter, 250, 50, 500, 500);

    workPlotter = new TimeSeriesSimulationPlotter("Work status", "");
    workPlotter.addSeries("employed", new MeanArrayFunction(collector.getEmploymentCS()));
    workPlotter.addSeries("non-employed", new
              MeanArrayFunction(collector.getNonEmploymentCS()));
    workPlotter.addSeries("retired", new MeanArrayFunction(collector.getRetiredCS()));
    workPlotter.addSeries("students", new MeanArrayFunction(collector.getInEducationCS()));
    GuiUtils.addWindow(workPlotter, 750, 50, 500, 500);

    eduPlotter = new TimeSeriesSimulationPlotter("Education level", "");
    eduPlotter.addSeries("low", new MeanArrayFunction(collector.getLowEducationCS()));
    eduPlotter.addSeries("mid", new MeanArrayFunction(collector.getMidEducationCS()));
    eduPlotter.addSeries("high", new MeanArrayFunction(collector.getHighEducationCS()));
    GuiUtils.addWindow(eduPlotter, 1250, 50, 500, 500);
}
```

Box 14. The *PersonsObserver.buildObjects*() method.

Other plots can be easily added. In particular, by building on the *JFreeChart* library, the *CollectionBarSimulationPlotter* class in JAS allows to create histograms for representing distributions of given variables in the simulated population, at any given simulation period.

The schedule of the *PersonsObserver* class simply manages the updating of these three plots. The *PersonsObserver* class also defines an extra parameter which controls the frequency of update of the plots, and which is loaded into the GUI:

```
@ModelParameter
private Integer displayFrequency = 1;
```

6.7 The StartPersons class

The *Start* class initialises the JAS simulation engine and defines the list of models to be used. In general, the Start class is designed to handle three types of experiments:

- performing a single run of the simulation in **interactive mode**, through the creation of a *Model* and related *Collectors* and *Observers*, with their GUIs;
- performing a single run of the simulation in **batch mode**, through the creation of the *Model* and possibly the *Collectors*; this involves managing parameter setup, model creation and execution directly, and is aimed at capturing only the simulation's numerical output;
- performing a **multi-run session** (whose structure is defined in a class extending the *MultiRun* interface) where the simulation is run repeatedly using different parameter values, so as to explore the space of solutions and produce sensitivity analyses on the specified parameters.

The Start class implements the *ExperimentBuilder* interface, which defines the *buildExperiment*() method. This method creates managers and adds them to the JAS engine. In *demo07*, the simulation is run in interactive mode (Box 15).

```
public class StartPersons implements ExperimentBuilder {
	public static void main(String[] args) {
		boolean showGui = true;

		StartPersons experimentBuilder = new StartPersons();
		final SimulationEngine engine = SimulationEngine.getInstance();
		MicrosimShell gui = null;
		if (showGui) {
			gui = new MicrosimShell(engine);
			gui.setVisible(true);
		}

		engine.setExperimentBuilder(experimentBuilder);

		engine.setup();

	}

	@Override
	public void buildExperiment(SimulationEngine engine) {
		PersonsModel model = new PersonsModel();
		PersonsCollector collector = new PersonsCollector(model);
		PersonsObserver observer = new PersonsObserver(model, collector);

		engine.addSimulationManager(model);
		engine.addSimulationManager(collector);
		engine.addSimulationManager(observer);
	}

}
```

Box 15. The StartPerson class.

The *Start* class contains the standard *main*() method which allows a Java application to run. By selecting the "run as Java application" option from the Eclipse menu, or by typing "java StartPersons" from the Command Prompt, this procedure launches the JAS GUI, creates a model instance and gives it to the simulation engine. It then creates a Collector and an Observer and calls the *setup*() method of the simulation engine, which has the task of loading the experiment into memory.

The JAS GUI contains a mask for setting the specific Model parameters, another mask for defining the specific Observer parameters and the three dynamic graphs defined in the *Observer* class. Figure 1 depicts the graphical output of one simulation run.

Figure 1. The graphical output of one simulation run.

The Tools > 'Database explorer' tab in the JAS GUI allows to browse the input and output databases. By selecting a specific database and pressing the 'Show database' button, the data can be explored in the default web browser using SQL commands. As an example, the output tables can be exported in CSV format for subsequent analysis using specific statistical tools by typing:

```
CALL CSVWRITE('person.csv', 'SELECT * FROM PERSON ');
```

7. Conclusions

The JAS simulation platform achieves a convergence between agent-based and microsimulation tools. The platform can be assessed both with respect to what it is, and with respect to what it is not. First, JAS is not a tool to speed up simulation execution; rather, its goal is to speed up model development, facilitate model documentation, and foster model testing and sharing. The rationale behind this choice lies in the observation that computer power is always increasing, while researchers' time is not. Also, large-scale simulation projects are generally beyond the reach of a single scientist. Even when they remain under the control of a restricted group of people, they generally require a prolonged effort, often on a stop-and-go basis. The possibility of building on work done in the past by the same authors or by other researchers is crucial. Simulation modeling needs cooperative development. The choice of an entirely open-source tool should also be evaluated in this light. Moreover, JAS does not force the user to adopt predefined solutions to the problems faced in simulation modeling. By offering a set of libraries that extend the capability of the standard Java classes, JAS leaves entirely open the possibility of using other libraries and tools, either as an alternative or on top of the JAS toolkit.In the trade-off between efficiency and transparency, JAS deliberately opted for the latter.

In the paper, we have described some of the potential of the platform, by developing a rich dynamic microsimulation with demographic and economic life-course events. We opted for replicating the most complete sample application available in LIAM2, another microsimulation platform, to allow a better evaluation of the JAS features. A more detailed (and quantitative) comparison of the two implementations, as well as a comparison of JAS with other simulation platforms such as ModGen, RePast, MASON and NetLogo is currently undergoing and will be available on JAS website.

References

Baekgaard H (2002). "Micro-macro linkage and the alignment of transition processes: some issues, techniques and examples". National Centre for Social and Economic Modelling (NATSEM) Technical paper No. 25.

Billari F, Prskawetz A (2003). *Agent-based computational demography*. Springer, Berlin.

De Menten G, Dekkers G, Bryon G, Liègeois P, O'Donoghue C (2014). "LIAM2: a New Open Source Development Tool for Discrete-Time Dynamic Microsimulation Models". *Journal of Artificial Societies and Social Simulation*, 17(3): art. 9.

Gilbert N, Terna P (2000). "How to build and use agent-based models in social science". *Mind & Society*, 1(1): 57-72.

Keller AM, Agarwal S, Jensen R (1993). "Enabling The Integration of Object Applications With Relational Databases". *ACM SIGMOD Proceedings.*

Klevmarken A (2002). "Statistical inference in micro-simulation models: incorporating external information". *Mathematics and Computers in Simulation*, 59: 255-265.

Li J, O'Donoghue C (2012). "Simulating Histories within Dynamic Microsimulation Model". *International Journal of Microsimulation*, 5(1): 52-76.

Li J, O'Donoghue C (2014). "Evaluating Binary Alignment Methods in Microsimulation Models". Journal of Artificial Societies and Social Simulation, 17(1): art. 15.

Luke S, Cioffi-Revilla C, Panait L, Sullivan K, Balan G (2005). "MASON: A Multiagent Simulation Environment". *Simulation*, 81(7): 517-527.

Luna F, Stefansson B (2000). *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*". Kluwer, Dordrecht.

Mannion O, Lay-Yee R, Wrapson W, Davis P, Pearson J (2012). "JAMSIM: a Microsimulation Modelling Policy Tool". *Journal of Artificial Societies and Social Simulation* 15(1): art. 8.

Minar N, Burkhart R, Langton C, Askenazi M (1996). "The Swarm simulation system: A toolkit for building multi-agent simulations". Santa Fe Institute Working Paper 96-06-042, Santa Fe.

North MJ, Collier NT, Ozik J, Tatara E, Altaweel M, Macal CM, Bragen M, Sydelko P (2013). "Complex Adaptive Systems Modeling with Repast Simphony". *Complex Adaptive Systems Modeling*, Springer, Heidelberg.

Richiardi M (2012). "Agent-based Computational Economics. A Short Introduction." *The Knowledge Engineering Review*, 27(2): 137-149.

Richiardi M (2013). "The missing link: AB models and dynamic microsimulation". In: Leitner S, Wall F (eds). *Artificial Economics and Self Organization*. Springer, Lecture Notes in Economics and Mathematical Systems, vol. 669, Berlin.

Stephensen P (2014). “An Information-Loss-Minimizing Approach to Multinominal Alignment in Microsimulation Models”. DREAM Working Paper 2014:4.

Statistics Canada (2009). “Modgen Version 10.1.0 Developer’s Guide”. Accessed on 17/12/2014 at http://www.statcan.gc.ca/microsimulation/pdf/dev-guide-eng.pdf.

Wilensky U (1999). “NetLogo (and NetLogo User Manual)”. Center for Connected Learning and Computer-Based Modeling, Northwestern University.